14 MAJOR AGILE FRAMEWORKS

This chapter does not pretend to describe all Agile frameworks in the market; rather this is a list of frameworks that are most popular in the market at the publication date. There are many other frameworks, many of which are named in the 'History of Agile' section (Section 1.1).

We define 'popular Agile frameworks' as the frameworks we have seen most widely used in the organisations we have interacted with over the last five years. All of the generic Agile practices that we have described in parts one to three of this book will be apparent within these frameworks.

14.1 EXTREME PROGRAMMING (XP)

14.1.1 The ethos of eXtreme Programming.

eXtreme Programming (XP; Beck, 2004) was conceived during the late 1990s to tackle the then-current methodologies' inability to deal with change. XP quickly became the dominant Agile methodology until Scrum overtook it in the early 2000s.

XP practices are based on a set of values and guiding principles; by understanding first the values, then the principles, teams can see why the XP practices are used. Without understanding the why, the practices lose their meaning and can easily slip.

XP takes proven practices and applies them in their purest, or most extreme form. It is a methodology for 'people coming together to develop software' and not specifically for programming (Beck, 2004).

There are many similarities between XP's values, principles and practices and the values of the Agile Manifesto. The now common Agile practices of stories, test-driven-development and continuous integration all originate from XP.

14.1.2 Values

Five values underpin the methods and behaviours applied in the principles and practices of XP.

14.1.2.1 Communication

Ambiguous or misinterpreted requirements, large unread documents, lack of face-to-face dialogue, isolated teams and other breakdowns in communication can lead directly to project delays or failure.

Communication is ingrained into XP – many of its practices cannot be done without communicating. Tasks such as planning and estimating must be performed together and are based on a verbal dialogue as opposed to a one-way document-based method. As a result of this approach, XP projects generally create less documentation than a Waterfall project.

XP teams are encouraged to communicate through practices such as co-location

and pair-programming (see Section 14.1.4.7). Unit tests describe the behaviour of individual units of the overall system, thus becoming a method of communication themselves.

14.1.2.2 Feedback

A process that embraces change must receive feedback whenever possible.

Feedback within XP happens at many levels. Working in short iterations, teams have the opportunity for regular customer feedback. The state of functional tests shows the current development state of the project, while unit tests feed back information about the state of the code-base against people's expectations.

By interpreting feedback and applying learning to the system, XP is able to adapt quickly to changes at all levels.

14.1.2.3 Simplicity

This value is neatly summed up by the phrase 'do the simplest thing that could possibly work'. This translates to focusing on solutions for the current iteration of work and contributes to the rapid development of stories.

Simplicity is difficult to apply, and the best intentions may still result in unfortunate consequences or an inability to deliver. For example, developers may be tempted to add features into the code they believe will be required in the future. XP teams focus only on what is needed right now. Extra functionality or features can lead to extra work to be completed, such as functional tests, unit tests or unforeseen consequences in other areas of the system.

14.1.2.4 Respect

Respect must underpin the other values for them to be effective. This is directly related to the Agile Manifesto statement 'Individuals and interactions over processes and tools' as without respect within the team there will not be focused and effective delivery.

By respecting oneself as well as the other team members, people are more likely to feel confident that they are making a valuable contribution to the team and project as a whole.

14.1.2.5 Courage

It can take courage to stick to the values of XP. For example, it takes courage to highlight issues that could have an impact on the project, such as an architectural flaw late in the day. It takes courage to throw away code when you recognise that there is a better design. It takes courage to refactor another developer's code. It takes courage to fail and it takes courage to change.

Without the other values, courage can become recklessness. Only by being respectful, focusing on current development and communicating openly are actions courageous.

14.1.3 XP principles

XP's principles translate the abstract values into concrete practices and should be used to guide a project during development and when selecting appropriate, alternative or even new practices.

14.1.3.1 Fundamental principles

Rapid feedback

Seek feedback at the earliest possible moment, interpret it appropriately and apply learning from it back into the system. In practice this is achieved by the different testing activities, direct communication with the customer and the sharing of knowledge across the whole team.

Assume simplicity

Choose the simplest solution that could solve the problem. By applying the principle of simplicity to development, design and code becomes leaner, resulting in quicker development. It is only at the point when a particular feature is actually needed, that its design and code will be tackled. The phrase 'You Ain't Gonna Need It' (YAGNI) was coined to embody this principle.

Applied blindly or in isolation, this principle can lead to a messy, hard-to-work-with system. Therefore it needs to be used alongside other practices such as refactoring, unit tests and continuous integration (see Section 8.10).

Incremental change

By working incrementally, teams focus on manageably sized tasks. Two-week iterations and their goals are much easier to focus on than a six month project.

By receiving regular and frequent feedback, many small, deliberate changes can be made. Large projects are broken up into small releases, difficult problems are broken up into a collection of smaller simpler problems and writing small tests creates code. It is even recommended that the introduction of XP practices themselves is applied in small manageable steps.

Embracing change

In software development change is a reality. Large, up-front plans and designs are costly to change. While this principle does not mean that all changes must be accepted, it encourages people to find new ways of working with change. So, for example, there will be no large up-front designs – instead teams will design as the project develops. Also, there will be no large requirements documents – teams will plan only as far as the next release. Essentially, this principle says that change is to be expected and that teams need to have a process that works with it.

Quality work

An XP team is committed to the principle of doing a good job. This can mean technically, or this can mean by keeping the core values and principles at the heart of what it does. By producing quality work, members of an XP team will be proud of their contribution to the project, which becomes a motivating factor.

Sacrificing quality will only have a negative effect on a project. As one of the

fundamental principles of XP, it should not be optional.

14.1.3.2 Further principles

These principles are more specific to particular situations.

- Teach learning: XP teams use their experience to apply XP practices (see below) to the appropriate degree. If some practices are less known, strategies are devised so team members can learn.
- Small initial investment: With a tight budget on a project, there is more likely to be a strong focus from both a requirements and technology perspective. Often this sort of environment can generate innovation.
- Play to win: The actions of individuals on an XP team are intended to help the team and project succeed or ‘win’, rather than to protect the individual from blame in the event of failure.
- Concrete experiments: Decisions should be underpinned by the result of experiments to reduce risk.
- Open and honest communication: Team members should be encouraged to communicate in an open and honest manner, even if it means delivering bad or uncomfortable news. Often, if team members do not feel able to do this, it is a sign of a larger cultural problem.
- Work with people’s instincts, not against them: The XP team trust their instincts to do the right thing.
- Accepted responsibility: Individuals are not told what to do, they take responsibility when required. This is part of self-organisation.
- Local adaptation: XP should be adapted to the respective environment. It may not be possible to follow all the practices exactly.
- Travel light: Unnecessary tools and practices should be avoided. Tools are tempting but they can also restrict project practices.
- Honest measurement: Metrics that are of no value to the team or project should be avoided, and metrics that are used should be made visible.

14.1.4 Practices

14.1.4.1 The planning game

The main planning activity in XP takes place at both release and iteration level.

In release planning, the customer will translate requirements into **user-stories** and the team will estimate how long each will take (**exploration phase**). Based on the business value, combined with the estimates, the customer will decide the scope and date of the next release (**commitment phase**).

While the exploration and commitment phases may be completed in a single whole-team meeting, the final phase of release planning, called the **steering phase**, is played over the remaining time to the release. In this phase, the project is guided, or **steered**, based on feedback from both the customer and development. During

the steering phase the team works in short iterations. Based on the user-story estimates and their business value, the customer selects enough stories to fill the iteration. The team then identifies the individual tasks required to deliver each story.

The team aims to deliver working software at the end of the iteration. Feedback from each iteration's delivery is used to **steer** the project. Both the customer and team have opportunities during the steering phase to make changes. The customer can add new user-stories, which receive a development estimate. In doing this, the customer must indicate which existing stories will be replaced.

The team can adjust their estimates. If the re-estimation is isolated in one story, then only the current iteration may be affected; however if the re-estimation has to be applied to the remaining user-stories, the release plan itself may be affected. In this case the scope or date of the release will need to be reassessed.

14.1.4.2 Small releases

XP teams should aim to create releases of valuable functionality as quickly as possible. It is the customer's responsibility to identify the functional release increments, moving away from the idea that the system cannot be released until complete.

By identifying small increments of functionality, releases can start to gather feedback that can be used in steering the system's subsequent development, as well as potentially delivering business value early.

14.1.4.3 Metaphor

By creating a metaphor, the whole team can form an understanding of the system, its elements and their relationships. To keep a metaphor relevant, its language should be used throughout the project. This allows the team to relate back to the metaphor, which is useful when describing technical changes. The metaphor may be used in place of, or as part of, the system architecture.

For example, many e-commerce applications have used the shopping cart as a metaphor to discuss requirements; another common e-commerce metaphor is to consider the product to be built using an auction metaphor (e.g. eBay).

14.1.4.4 Simple design

The design applied by the developers must be as simple as possible, satisfying only the requirements for the functionality being implemented.

The design must still satisfy any relevant quality criteria or standards, and it may require a number of refactoring exercises (see Section 8.10.1) before the simplest design reveals itself, but it means the code will be easier to understand, test and modify.

14.1.4.5 Testing

All **stories** are to have automated tests (see Section 8.10).

Automated functional tests, with criteria specified as part of each user-story,

provide the customer with two key benefits:

1. Indications of progress as new tests are shown to be successful.
2. Confidence in the system as existing tests are shown to be successful.

The team will be driven by tests. Test Driven Development is the practice of writing a unit test before creating the code itself.

14.1.4.6 Refactoring

Refactoring is the process of simplifying the internal structure of code without affecting its external behaviour (see Section 8.10.1).

When used in combination with test-driven development, developers iterate code once a test is written to arrive at the simplest possible solution. The unit-test verifies that the code still operates as expected.

14.1.4.7 Pair programming

Code is created by **two** developers using **one** machine. While the developer at the keyboard is focused on creating the current test and code, the other is thinking from a different perspective – for example, they may be considering how the code fits into the overall solution or thinking of different test scenarios. After a period of time or at a convenient point, the developers swap places.

Pair programming has a number of benefits:

- Conversation during the process helps to quickly move the solution on.
- Knowledge is shared as developers pair with different individuals.
- Code is reviewed in real-time; teams that practise pair programming often eschew code reviews.
- The practice promotes collective code ownership.

14.1.4.8 Collective ownership

Developers can improve any part of the code at any time.

When performed in concert with pair programming and good testing, knowledge of the system is shared very quickly. This practice also avoids code becoming owned by individuals, which can lead to bottlenecks in development and poorly designed code.

14.1.4.9 Continuous integration

The codebase should be integrated and automated tests run frequently.

Developers working locally on their machines should check-in their changes frequently, ensuring that code conflicts are identified and resolved quickly.

In tandem with this, the codebase should be integrated automatically each time a change is checked in. On a successful build of the system, automated tests should be executed. The results of both the build and the test run should be clearly visible,

with any failures acted on immediately.

14.1.4.10 Forty-hour week

To ensure that team members remain creative, enthusiastic and perform at their best, XP teams must aim to work at a sustainable pace. XP does not forbid overtime, but it has a clear rule – *You can't work a second week of overtime.* The need for overtime should be mitigated by the other practices of XP; in particular the feedback aspects should allow problems to be discovered and addressed early rather than uncovered late in the day. The constant need for overtime on an XP project is an indication that estimates may need re-examining (see Section 14.1.4.1*)*.

14.1.4.11 On-site customer

A real customer should sit with the team. This person will be someone who will use the system, who has the knowledge and authority to answer questions and who can provide business related clarification so that issues don't block the progress of the iteration. It is expected that the customer will still carry out their normal day-to-day role, just co-located with the team.

14.1.4.12 Coding standards

A common coding standard, agreed by all developers, must be adopted across the team. The standard should refer to the other practices, such as simple design, and where possible identify established coding principles such as the SOLID principles (Martin, 2000). As the project progresses, the code will become identified by team coding style as opposed to any individual's coding style. This benefits the whole team, removing one of the barriers to **collective ownership**.

14.2 SCRUM

According to the State of Agile Development Survey (VersionOne, n.d.), Scrum or various Scrum variants are the most popular Agile frameworks, achieving 72 per cent. This section provides a more detailed approach of the most significant practices.

14.2.1 Overview

The Scrum framework (Schwaber and Sutherland, n.d.), or simply Scrum, is an iterative and incremental Agile framework, which is based on the three pillars of empirical process control, Transparency, Inspection and Adaptation (see Section 2.6.1):

- Transparency: provide visibility to all stakeholders, customers and anyone else responsible for the outcome.
- Inspection: check in a timely fashion on how well a product is progressing towards its goals.
- Adaptation: adjust the process to minimise deviation from those goals.

The Scrum framework consists of a set of values, roles, activities and artefacts that

form a holistic approach to delivering products. The people-centric framework adjusts to the characteristics of each environment, making every implementation unique within IT and non-IT-related environments. The first public announcement of Scrum (presented as **methodology**) took place in the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) conference in 1995, by its originators Jeff Sutherland and Ken Schwaber (Sutherland and Schwaber, 2013).

14.2.2 Scrum roles

A Scrum team has three roles with specific properties and responsibilities:

- ScrumMaster.
- Product owner.
- Development team.

The ScrumMaster facilitates the adoption of Scrum through continuous coaching and guidance, leading the Scrum team to high-performance and using the framework to inspect and adapt. The product owner defines what will be delivered in what order (via the product backlog), and the development team defines how to deliver what has been asked for and how long it will take (via the sprint backlog). The development team is a self-organising, cross-functional and collaborative entity, which aims to deliver the goals agreed with the product owner.

14.2.2.1 ScrumMaster

The ultimate ownership of Scrum values, principles and practices lies with the ScrumMaster. The servant-leadership approach (see Section 6.3) differentiates this role from traditional project management and development management roles, exercising no control and authority over the team (Adkins, 2010).

The ScrumMaster acts in a mentoring and coaching capacity similar to a change agent, providing process leadership and facilitating the use of Scrum at the team level to enable (amongst others) self-organisation and high-performance skills. In addition, the ScrumMaster also works at the organisation level, mitigating impediments towards the adoption of an organisation-specific Scrum approach.

Through specific activities and continuous coaching, the ScrumMaster helps the development team overcome impediments, resolve issues of internal or external nature and make best use of the lightweight and flexible framework. Blocking issues beyond the remit of the development team are addressed and escalated by the ScrumMaster, allowing the team to concentrate on the sprint goal. External interference/noise should be eliminated, as the ScrumMaster acts as a team armour in line with Agile Manifesto principle number five: *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done*.

A ScrumMaster is like a sheepdog, guiding and shepherding the team.

By adopting a continuous improvement mentality, the ScrumMaster challenges

norms of development that inhibit performance, while maintaining constant focus on the goal of the current objectives in line with Agile Manifesto principle number 12: *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts accordingly*.

In summary, the ScrumMaster is responsible for the following:

- The success of the Scrum process.
- Establishing Scrum practices and rules, shielding the team and removing obstacles.
- Ensuring that the Scrum team is fully functional and productive.
- Enabling close cooperation across all roles and functions.
- Ensuring that the Scrum process is followed, including effective daily Scrum meetings, sprint reviews, retrospectives and planning meetings.
- Leading and coaching the organisation in adopting Scrum.
- Assisting the product owner and finding effective product backlog management techniques.

14.2.2.2 Product owner

The product owner is the ‘empowered central point of product leadership’ within a Scrum team (Rubin, 2013). The main responsibilities of the role are defining and prioritising Product Backlog Items (PBIs) as well as maximising the value of the product overall.

Communication boundaries should not exist in a Scrum team, and such decisions are not made in isolation: requirement elaboration and prioritisation is supported by frequent contributions from the development team. Being the main liaison between the development team and the rest of the organisation, product owners are the ‘single voice of the customer’, collaborating with product managers, business analysts, customers and other stakeholders to determine requirements (Schwaber, 2004). It falls within their remit to maintain and effectively communicate a clear vision and set of objectives for the sprint and the release to all participants.

In addition, product owners have outward-looking activities (e.g. stakeholder management), including participating in strategic meetings, portfolio management sessions and cross-departmental discovery exercises. Despite their busy schedule, product owners must be readily available to answer questions on a just-in-time basis and steer the development team to the right direction through face-to-face communication.

Here is a summary of activities for the product owner:

- Identifying relative value on the PBIs.
- Communicating the vision of the business to the team and the vision of the team to the business.
- Defining available budget.

- Setting goals for the sprints and releases.
- Participating in the sprint planning and release planning meetings.
- Elaborating PBIs on a just-in-time basis with the team.
- Accepting PBIs.
- Accepting the sprint/release.
- Deciding when to release.
- Defining the features of the product via PBIs (mainly stories (see Section 7.1))
- Setting development schedule by ordering the product backlog.
- Adjusting PBIs and prioritising every sprint as needed.
- Ensuring return on investment.
- Gaining insight and assurance the product is meeting its goals through deep and broad feedback.

14.2.2.3 Development team

The nature of the development team in Scrum is fundamentally different from teams in traditional, Waterfall and command-and-control settings. In such models, existing organisational structures promote functional silos, communication barriers and wasteful handovers.

The diverse and cross-functional collection of specialisations relies on a matrix organisational structure that potentially comprises coders, testers, architects, product managers, business analysts, user experience designers, other specialist and supporting personnel, and so forth – basically everyone required to get the PBI to 'done' status.

Self-organisation, a critical aspect in Scrum, is an evolving characteristic of the development team. The team is allowed to determine the best manner to realise the requirements that the product owner defined and prioritised in the form of sprint goal in line with Agile Manifesto principle number 11: *The best architectures, requirements, and designs emerge from self-organising teams*. According to the *Scrum guide* (Sutherland and Schwaber, 2013), the ideal team size should be small enough to maintain sufficient collaboration and communication levels, and large enough to complete significant, high-quality, working software items within given sprints (Agile Manifesto principle number 7: *Working software is the primary measure of progress*). The typical number of members in a development team spans from 5 to 9 members (rule of thumb: 7 ± 2 developers). Smaller teams are likely to experience skill constraints, inhibiting the delivery of potentially releasable increments. On the other hand, larger groups are unable to manage the complexity introduced for empirical methods.

The responsibilities of the development team are summarised below:

- Working with the product owner, ensuring that PBIs are understood and realised appropriately.

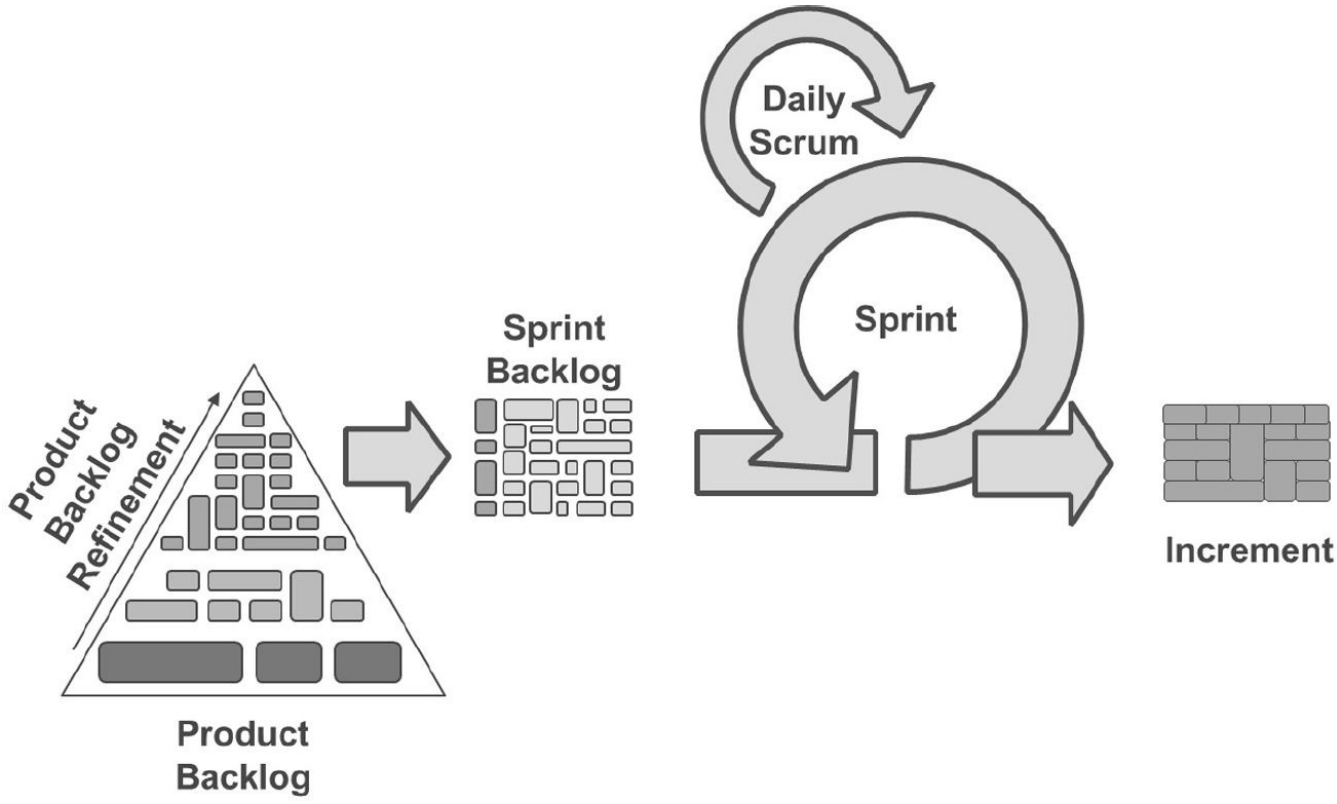
- Defining emergent architectures while maintaining quality at the agreed level.
- Self-organising, cross-functional with no predetermined roles.
- Seven plus or minus two people – all skills required to get PBIs to ‘done’.
- Lots of face-to-face communication.
- Responsible for organising tasks and committing to work.
- Authority to do whatever is needed to meet commitment.
- Demonstrates work results to the product owner and stakeholders.
- Has the right to do everything within the boundaries of the delivery standards and guidelines to reach the sprint/release goal.

14.2.3 Scrum activities and artefacts

14.2.3.1 Scrum framework

This most popular and lightweight Agile framework follows a particular process (see Figure 14.1), which is outlined in the following paragraphs.

Figure 14.1 Scrum process



Traditionally, the strategic intent of the delivery is communicated through the product owner to the Scrum team. Product backlog items may include critical contextual information and supplementary documentation. As the development effort required to realise all PBIs in the product vision can be large, the Scrum team engages in a decomposition and prioritisation activity, called refinement or grooming, which breaks down items into a number of smaller, clearer and more concise items that are deemed ‘ready’ (see Section 10.2) for sprint planning, which the development team can turn into product increments, ideally within two to five days. The team will refine the PBIs guided by prioritisation. The product backlog

must be prioritised, ordered and estimated, as it is used for planning and forecasting future sprints or releases. Items further down the product backlog will be less well understood and therefore estimates will be less precise.

Development time is divided into time-boxes with the duration of no more than one month, with a preference for a shorter timescale known as **sprints**. Each sprint starts with a planning exercise, named **sprint planning**, which determines the set of prioritised PBIs that the development team is likely to realise in a collaborative manner. The team plan how they will develop the product increment from the PBIs, and create a sprint backlog, which contains the known tasks required to create an increment for each PBI planned for the coming sprint. Nevertheless, changes should be expected as the team goes through the 'cone of uncertainty' (see Section 2.5) and more information surfaces.

It is imperative that the team feels comfortable with the items listed in the sprint backlog and commits to its execution to the best of their abilities. Commitment is an essential element of Scrum, as the development team commits to the sprint goal which is typically a subset of the PBIs in the sprint, the sprint backlog is always a forecast, the sprint goal is a commitment. Throughout the duration of the sprint, the development team focuses on PBI design, implementation, integration, testing and documentation (sprint execution), which concludes with two inspect-and-adapt events: the sprint review and sprint retrospective.

In the sprint review, which is the public end to the sprint, the Scrum team demonstrates the implemented PBIs to the business participants and gathers direct feedback. During the sprint retrospective, the 'private end to the sprint', the team pursues a process improvement exercise, identifying areas for improvement. The outcome from the session should be an actionable list of improvements, which should be part of the subsequent sprint backlog.

14.2.3.2 Product backlog

The fundamental focus of Scrum on early and continuous delivery of valuable software depends partly on a Lean and transparent requirements model known as the product backlog, which supports the demands of external stakeholders and team members (see Agile Manifesto principle number 1: *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software*). The product owner, with assistance from the ScrumMaster, Scrum team and business stakeholders, is ultimately responsible for defining, elaborating, sequencing and communicating added-value work in an ordered fashion. Work items are sorted in descending order, with relatively higher value items appearing at the top and lower value items appearing towards the bottom. The process of eliciting and ranking requirements is called value-based decomposition and prioritisation.

Product backlog items may be defined at differing levels of clarity and understanding as they are refined prior to sprint planning, incorporating changes based on feedback and time-boxed investigative work ('spikes' – see Section 7.1.3).

As mentioned before, requirements definition and progressive elaboration does not occur in isolation. The product owner liaises with internal and external stakeholders to define the PBIs. Regardless of the prioritisation scheme used, all stakeholders should comprehend the factors associated with the current sequence, such as relative business value, risk involved, available knowledge, return of investment, expected monetary value, and so on. The evolution of the product backlog, known as backlog refinement (also backlog grooming), is an activity that takes place very frequently within Scrum teams. PBIs are refined (broken down), added, removed, modified and re-prioritised according to the constantly changing business needs and competitive landscape. Throughout the course of development, the product owner and the development team gather useful information via feedback loops (see 'sprint review' below), which has impact on the PBIs and their priority.

14.2.3.3 Sprints

Product development work (or any type of work in non-IT environments) in Scrum is performed in time-boxed (time-limited) iterations or cycles, producing a potentially releasable increment that contributes to the value stream.

The suggested time-box should be a maximum of a month with a preference for a shorter timescale. Most development teams start at two weeks and then change from that length if required. Although the duration of all sprints should remain constant, it could be revisited periodically, as a part of continuous improvement discussions between the ScrumMaster and the development team. Each sprint is followed by another sprint, forming a cadence, which is considered the 'Agile' heartbeat used to coordinate various Scrum-related events and activities, such as sprint planning meetings, sprint reviews, retrospectives, and other non-Scrum-related tasks.

Each sprint has a sprint goal, meaning a set of objectives mutually agreed between the development team and the product owner. Fundamentally, altering the goal mid-sprint is not an acceptable practice, as changes in sprint scope disorientate the Scrum team.

14.2.3.4 Sprint planning

Prior to any development, each sprint starts with a session when team plan their work. During this significant planning activity, known as the sprint planning meeting, the Scrum team accumulates knowledge in a collaborative manner and the product owner defines most added-value PBIs from the product backlog. As the Scrum team agrees, the selected work items are scheduled for execution.

Information sharing, conversations and other collaborative activities should be specific to planning. Therefore, it is common practice that sprint planning follows a predetermined agenda.

Within 'sprint planning part 1' the product owner initially defines the set of achievements, named the 'sprint goal', which should be accomplished at the end of the sprint, in agreement with the team. This high-level goal acts as a driver of the conversations and decisions that will follow. The development team commit to the

sprint goal, the product owner commits to not add PBIs or any other noise to the development team during the sprint so they can focus on delivering the sprint goal.

Many development teams proceed into further decomposition of the chosen product backlog items into technical tasks, this is known as ‘sprint planning part 2’. Through conversations, team members have better visibility and understanding of the effort needed to realise the PBI at hand. This approach achieves just-in-time planning that eliminates wasteful up-front discussions. Development teams do sufficient design and planning to be confident of completing the work in the sprint. Work to be done in the early days is broken down into units of one day or less.

The collection of PBIs chosen, including their technical tasks and estimations, make up the **sprint backlog**, which will change throughout the course of the sprint as the team update their estimates. The team own the sprint backlog and must maintain it throughout the sprint, guiding the development team in their pursuit of the sprint goal.

14.2.3.5 Sprinting

Following the commitment of the development team to the mutually agreed sprint goal, sprint execution starts. The main focus of the development team during delivery, facilitated and coached by the ScrumMaster, is to implement the product backlog items through tasks.

The development team is empowered to define task-level work without interference and follows the necessary steps to get the PBIs ‘done’ (see Section 10.2.1).

14.2.3.6 Daily Scrum

Face-to-face communication and synchronisation meetings are essentials in Agile. Similarly in Scrum, the development team participates in a recurring meeting, which takes place on a daily basis at an agreed time. Known as the daily Scrum, it is a strictly time-boxed session, up to 15 minutes, which allows the team members to understand current progress, synchronise activities and share information about obstacles hindering progress towards the sprint goal. Daily Scrums have strong roots in ‘Kaizen’ culture (continuous improvement – Lean) and are frequently referred to as daily stand-ups, as team members usually remain standing throughout its duration, in order to keep conversations concise and the time short.

The standard agenda of a daily Scrum, which is facilitated by the ScrumMaster, consists of three straightforward questions, which are answered by each participant:

- What did I do yesterday?
- What do I plan to do today?
- What is getting in my way?

The ‘what did I do yesterday?’ question gives the opportunity to every participant to outline what they have accomplished since the last daily Scrum. This is followed

by the ‘what do I plan to do today?’ question, which provides visibility of what the next steps are to the rest of the team and encourages a commitment between peers. The combination of the first two questions highlights a sequence of events that are important for all team members to synchronise. Finally, the ‘what is getting in my way?’ question allows members to raise concerns about current progress, brings impediments to the attention of the team and allows for modifications to anything that is impeding progress should these be considered necessary. The ScrumMaster plays a pivotal role in removing any obstacles to moving forward.

According to Jean Tabaka (2006), the questions above could be posed differently, emphasising the commitment amongst team members:

- What did I commit to do for all of us yesterday?
- What am I going to commit to do for the team today?
- What may be preventing me from meeting my commitment to the team till the next daily scrum?

To further focus on the benefits of the actionable status information, Craig Larman (2003) suggested two additional questions for daily Scrums:

- Have any additional tasks been identified for this sprint (in the sprint backlog)?
- Have you learnt or decided anything new which would be of relevance to some of the team members?

Daily Scrums improve communication and provide an excellent platform for adaptive daily planning and self-organisation. As they significantly contribute to the progression of development effort, participation is anticipated. Nevertheless, to keep the meetings efficient, yet ensure that communication flows effectively, the rules of the daily Scrum distinguish which members should actively participate from those that solely observe. Daily Scrums are open to stakeholders outside the strict boundaries of the Scrum team; however, they observe as listeners rather than contributors. The Scrum team, including the product owner (if the team have agreed the product owner will be present), must participate as they are all committed to the sprint goal.

14.2.3.7 Definition of done

See Section 10.2.1.

14.2.3.8 Sprint review

In the course of each sprint, the development team adopts an inward-looking approach towards development, focusing on achieving the sprint goal through commitment and self-organisation. At the finish line, each sprint concludes with two additional inspect-and-adapt activities: the sprint review and the sprint retrospective. Particularly in sprint review, the team has an excellent platform to engage in inspect-and-adapt conversations amongst its members and with the rest of the organisation.

The inspect-and-adapt activity consists of presentations/reviews of the completed sprint backlog items within the context of the product vision and future development scope. The current status of development effort becomes apparent to all participants, allowing everyone to steer the scope of future sprints to the right business direction. The bidirectional nature of information flow improves visibility between the Scrum team and business stakeholders (Rubin, 2013). The former has better clarity and appreciation of the business requirements through direct feedback, whereas the latter understand the work ‘done’ in the potentially shippable increment. Sprint reviews are also an opportunity for the Scrum team to showcase their technical achievements, which frequently has a positive effect on team morale and empowerment.

14.2.3.9 Sprint retrospective

In Agile environments, where continuous improvement and Kaizen culture thrive, sprint retrospectives are the activities which celebrate process optimisation, team building and collaboration building activities. The last ceremony in Scrum – before the end of the sprint – is essential in identifying opportunities for performance and collaboration improvements, deciding on adjustments of the current delivery approach and understanding any knowledge and skill-set gaps. It also strengthens the bond amongst the Scrum team members, who ‘continue their path from divergence to convergence’ (Tabaka, 2006).

As the Scrum team reflects and identifies opportunities for improvement, the ScrumMaster facilitates the process of engaging the Scrum team to commit to such process improvement actions. All action points should be undertaken during the following sprint.

14.3 DSDM

DSDM is an Agile project management and delivery framework from the DSDM Consortium (DSDM Consortium, 2014b), of which the Agile project framework is the latest version. In the DSDM Consortium’s own words:

> We’re all about scalable Agile delivery of projects and programmes with the right amount of governance and control that enables successful innovation.
>
> (DSDM Consortium, 2014b)

Many other Agile frameworks describe an approach that works well for delivery of a product, but they do not provide the governance that is required when working within a project governance structure. DSDM provides the governance necessary to run an effective Agile project and deliver the product. DSDM is also equally applicable to IT and non-IT projects (by replacing the ‘working software above comprehensive documentation’ line on the Agile Manifesto with ‘working **solution** above comprehensive documentation’.

DSDM defines a philosophy that is supported by eight principles, a development process, roles and responsibilities and a comprehensive set of products, all of which are supported by five practices.

14.3.1 The DSDM philosophy and eight principles

The DSDM philosophy is that:

> … best business value emerges when projects are aligned to clear business goals, deliver frequently and involve the collaboration of motivated and empowered people.
>
> (DSDM Consortium, 2014b)

The philosophy is supported by the eight guiding principles; these provide the mind-set and behaviours needed for a project to be successful.

Principle 1 – Focus on the business need Deliver what the business wants when the business needs it.

Principle 2 – Deliver on time By working in time-boxes and applying MSCW the team will always make a delivery on time.

Principle 3 – Collaborate Teams who collaborate will outperform those who don't. Collaboration increases team understanding, improves speed to deliver and creates a sense of shared ownership.

Principle 4 – Never compromise on quality Aim to deliver the agreed level of quality from the beginning of the project through to the end.

Principle 5 – Build incrementally from firm foundations Building from firm foundations is critical to be able to deliver value early and continue to add value throughout a project. Firm foundations are created by doing the appropriate level of investigation up front – enough design up front (EDUF), rather than no design up front (NDUF) or big design up front (BDUF). This is particularly important for large or complex projects/programmes or in large or complex organisations.

Principle 6 – Develop iteratively Iterative development allows teams to converge on the best solution by using a cycle of Thought, Action and Conversation.

Principle 7 – Communicate continuously and clearly Poor communication is often the cause of project failure. DSDM uses many processes and practices to support effective communication.

Principle 8 – Demonstrate control Any project management methodology needs to be in control, as do all roles working on an Agile project. DSDM demonstrates control by involving the whole team in creation of plans which are visible to all, and then by measuring progress through focus on delivery of products, rather than completed activities.

14.3.2 The DSDM roles and responsibilities

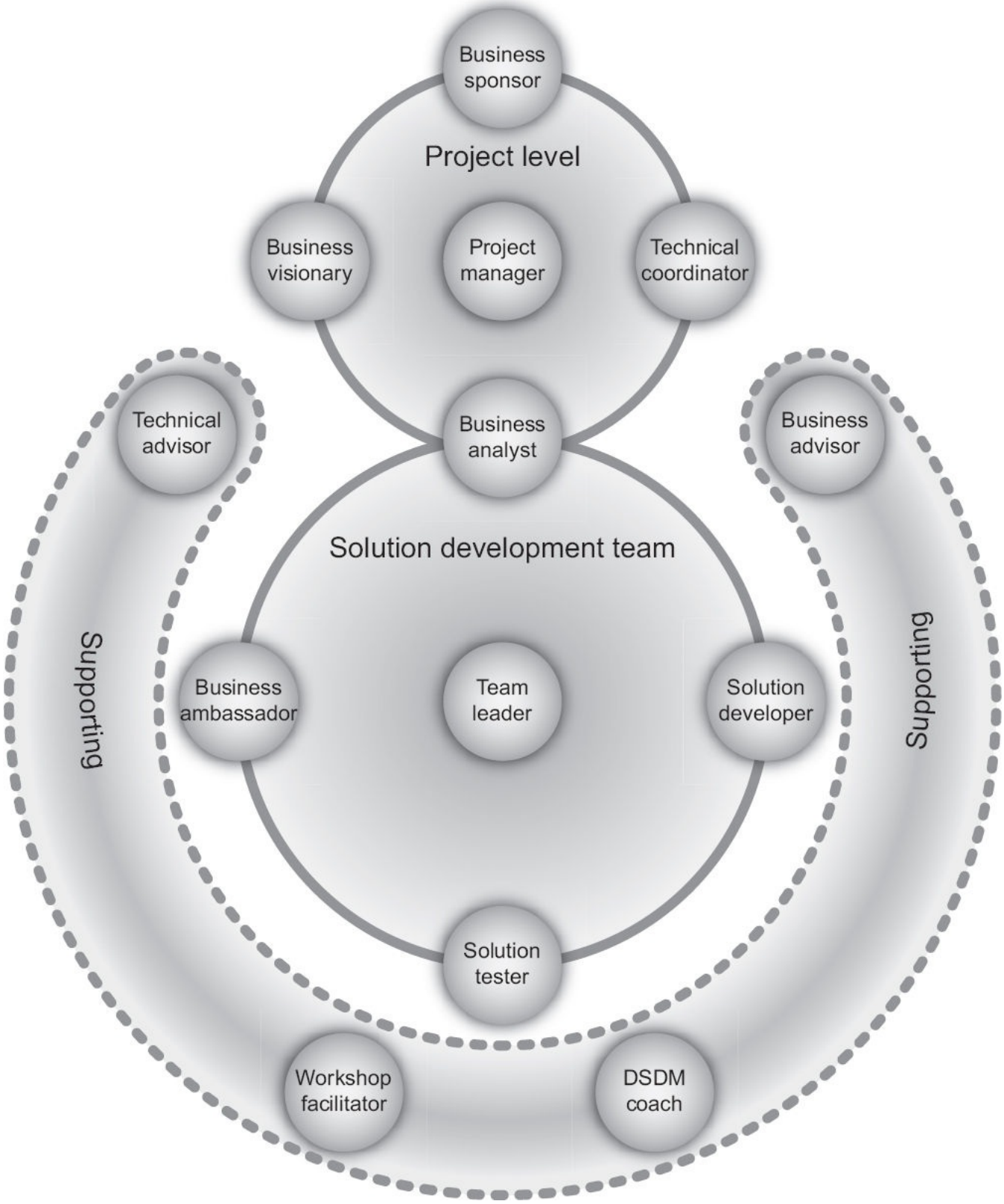
DSDM provides a fuller set of roles than other Agile frameworks, to reflect the additional complexity of a project environment. The roles are split into three main groups, project level, solution development team level and supporting level (see

Figure 14.2). A person can take more than one role, and a role can be covered by more than one person. It is therefore important to identify and communicate who will be taking responsibility for which roles.

The project level roles are:

- **Business sponsor** The business sponsor is the person who is funding the project.
- **Business visionary** The business visionary provides the team with the 'big picture' – the context and strategic direction to ensure the solution delivers benefit to the business. This role should be held by a single individual, since a project needs a single clear vision to avoid confusion and misdirection.
- **Project manager** The project manager coordinates all aspects of management of the project at a high level, but in line with the DSDM concept of empowerment, the project manager is expected to leave the detailed planning of the actual delivery of the product(s) to the members of the solution development team.
- **Technical coordinator** As the project's technical authority, the technical coordinator ensures that the solution/technical roles work in a consistent way, that the project is technically coherent and meets the desired technical quality standards.

Figure 14.2 DSDM roles



The solution development team roles are:

- **Business ambassador** The business ambassador is a key role. During iterative development, the business ambassador is the main day-to-day decision maker on behalf of the business. For this reason the business ambassador needs to be someone who is respected by their business peers and who has sufficient seniority, empowerment and credibility to make decisions on behalf of the business, in terms of ensuring the evolving solution is fit-for-business-purpose.

- **Business analyst** The business analyst is both active in supporting the project-level roles and fully integrated with the solution development team. The business analyst facilitates the relationship between the business and technical roles, ensuring accurate and appropriate decisions are made on the evolving solution on a day-to-day basis.
- **Team leader** The team leader is the servant-leader for the team, making sure the team follow their development process and removing impediments for the team. The team leader should foster a self-organising team.
- **Solution developer** The solution developer collaborates with the other solution development team roles to interpret business requirements and translate them into a solution that meets functional and non-functional needs.
- **Solution tester** The solution tester is fully integrated with the team and throughout the project performs testing in accordance with the agreed strategy for technical testing.

There are four supporting roles:

- **Technical advisor** The technical advisor supports the team by providing specific and often specialist technical input to the project from the perspective of those responsible for operational change management, operational support, ongoing maintenance of the solution etc.
- **Business advisor** The business advisor provides specific and, often, specialist input to solution development or solution testing – a business subject matter expert. The business advisor will normally be an intended user or beneficiary of the solution or may be a representative of a focus group.
- **Workshop facilitator** The workshop facilitator is responsible for managing the workshop process and is the catalyst for preparation and communication. The facilitator is responsible for the workshop process, organising and facilitating a session that allows the participants to achieve the workshop objective.
- **DSDM coach** Where a team has limited experience of using DSDM, the role of the DSDM coach is key to helping team members to get the most out of the approach, within the context and constraints of the wider organisation in which they work.

14.3.3 The DSDM process

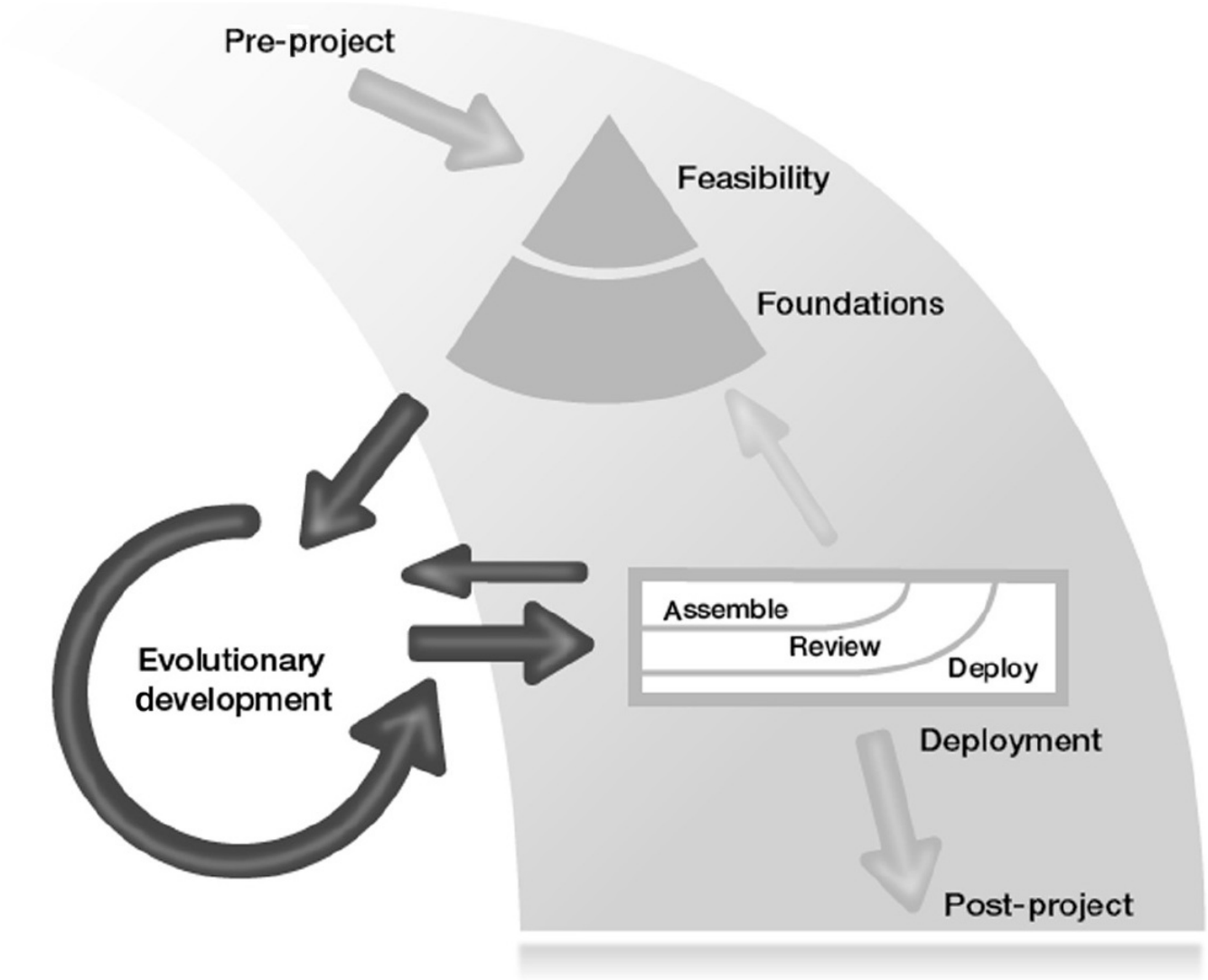
The DSDM process has several phases (see Figure 14.3).

- **Pre-project** The pre-project phase is a very short one, where the idea for a project is formed. Depending on the project and organisation this may be a formal terms-of-reference document, or something as simple as an email.
- **Feasibility** The feasibility phase takes the information from the pre-project phase and starts to create outline requirements, potential solution(s) at a high level, an outline plan and an outline business case. Just enough needs to be

done to establish if the project is feasible.

- **Foundations** The foundations phase takes the output from the feasibility phase and adds enough detail to the requirements, solution and plan for there to be a sufficiently firm foundation for the project to go into development.

Figure 14.3 DSDM process



The pre-project, feasibility and foundations phases are sequential. The business case is reassessed at each point to ensure the project is still good to go ahead.

- **Evolutionary development** Building on the firm foundations that have been established for the project, the evolutionary development phase evolves the solution, working within time-boxes. The solution development team(s) apply practices such as iterative development, time-boxing, and MSCW prioritisation, together with modelling and facilitated workshops, to converge on an accurate solution that meets the business need and is also built in the right way from a technical viewpoint.
- **Deployment** The objective of the deployment phase is to release one or more increments of the solution into operational use. Each deployment may be the final solution or a subset of the final solution. The ongoing project viability should also be assessed. If this is the last deployment, the project will be

formally closed.

- **Post-project** Post-project is the assessment of whether the solution has delivered the expected benefits. This means it may need to happen 3 to 6 months after deployment so that the benefits can start to emerge. The business sponsor and business visionary are responsible for validating that the business benefits have been received.

14.3.4 The products (a summary)

DSDM defines a complete set of products that can be used while managing an Agile project delivery. It is often not necessary to use all of the products defined by DSDM; instead it is better to tailor the products into the leanest possible subset.

The products can be categorised into three types:

- **Business products** – prioritised requirements, business vision, business case etc.
- **Management products** – plans, management foundations, control information e.g. risk log and review records etc.
- **Solution products** – the solution itself, supported by lean and timely documentation such as test records, an up-to-date definition of the solution architecture definition.

DSDM provides an explanation of when in the project delivery lifecycle each project should be created and updated, and additional detail on the purpose and constituent parts of each product.

14.3.5 The five practices

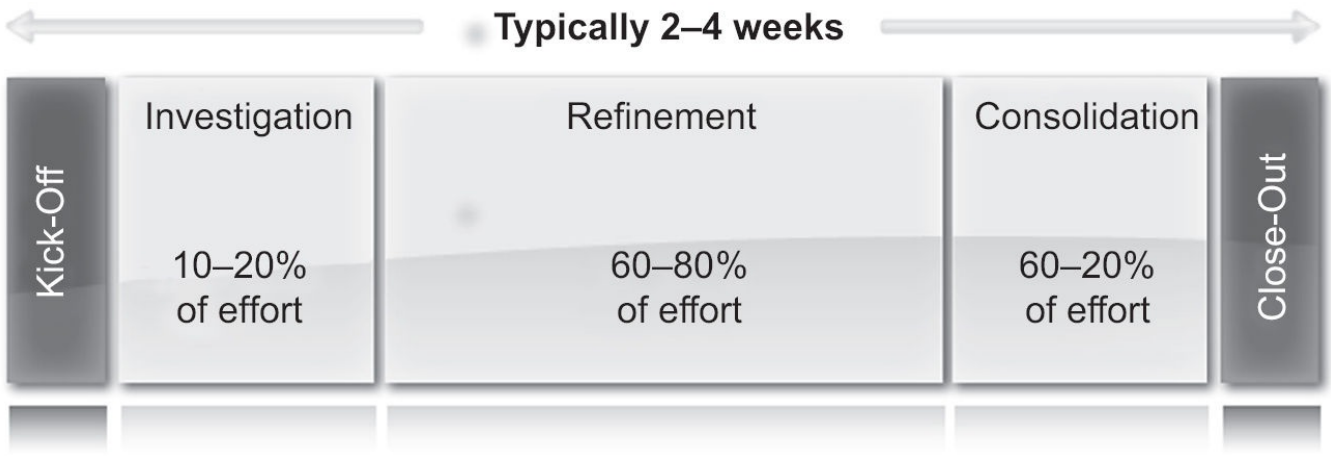
DSDM identifies five practices that help make a project delivery successful:

- **MSCW** – is a requirements prioritisation technique used in DSDM (see Section 7.1.4). MSCW stands for Must have, Should have, Could have, Won't have (this time). Ensuring the right balance of MSCW priorities helps build successful predictable projects.
- **Facilitated workshops** – use an interactive workshop environment, effective group dynamics and visual aids to extract high quality information in a compressed time frame in order to meet a predetermined set of deliverables. Workshops help build a collaborative working environment and group consensus, an essential part of Agile. It is good practice to have an independent facilitator who has no stake in the outcome of the workshop. The facilitator's role is to manage the workshop process, which then allows the participants to focus on providing high quality content, based on their personal knowledge and expertise in the subject being discussed. A well-organised facilitator will send out workshop information in advance, especially where participants need to do some pre-work or investigation before the workshop. After the workshop it is important to communicate any

output to all attendees.

- **Modelling** – is another practice that helps build effective communication. 'A picture is worth a thousand words.' Creating a model, whether this is a simple drawing on a flip chart, a process flow, a scaled model, or even a prototype, allows people to see what is being discussed or proposed. Models allow people to visualise a solution far better than reading a list of requirements, and this in turn allows people to create or challenge ideas.
- **Time-boxing** – most Agile methods would describe time-boxing as a simple fixed period of time to develop the product. DSDM, however, focuses more detail around time-boxing and recognises two different styles – a free-format style and a DSDM structured time-box. For both styles, DSDM recommends a fixed length of time, typically 2–4 weeks, and both styles are topped with a kick-off stage and tailed with a close-out stage (see Figure 14.4).

Figure 14.4 The DSDM structured (three iteration) time-box



A DSDM structured time-box can be helpful where the business availability is limited, as the structure is very useful to allow forward planning of the times when the business ambassador will attend specific planning, feedback and review sessions.

- **Kick-off** (1–3 hours) is a short session at the beginning of a time-box to understand the time-box objectives.
- **Investigation** (10–20 per cent of time-box) is where the team look at the requirements and define what work needs to be done to meet the objective.
- **Refinement** (60–80 per cent of time-box) is where the product is developed.
- **Consolidation** (10–20 per cent of time-box) is where the team double-check everything is complete and meets all acceptance criteria.
- **Close-out** (1–3 hours) is where the project visionary and technical coordinator formally accept the product.

Investigation, refinement and consolidation each complete with a review.

The free format time-box (Figure 14.5) also starts with a kick-off and finishes with a close-out. However, in between, there may be any number of formal or informal

review points. Typically the team pick up one or more products or user stories and evolve these iteratively until completed. Completion means a product or user story meets the previously agreed acceptance criteria. The team then pick up the next product or user stories and repeat the process. This free-format style relies on consistent business availability to review and provide feedback on an ongoing basis.

Figure 14.5 The DSDM free-format time-box

- **Iterative development** – is a process in which the evolving solution, or a part of it, evolves from a high-level concept to something with acknowledged business value. Each cycle of the process is intended to bring the part of the solution being worked on closer to completion and is always a collaborative process, typically involving two or more members of the solution development team. Each cycle should:
    - Be as short as possible, typically taking a day or two, with several cycles happening within a time-box.
    - Be only as formal as it needs to be – in most cases limited to an informal cycle of thought, action and conversation.
    - Involve the appropriate members of the solution development team relevant to the work being done. At its simplest, this could be, for example, a solution developer and a business ambassador working together, or it could need involvement from the whole solution development team, including several business advisors.

14.4 AGILE PROJECT MANAGEMENT

Agile project management (AgilePM; DSDM Consortium, 2014a) is based on the current version of DSDM (see Section 14.3). AgilePM provides a view of DSDM from the point of view of a project manager, managing an Agile project using an appropriate Agile management style i.e. facilitative, rather than command and control. AgilePM focuses on what an Agile project manager needs to consider and the behaviours and mind-set an Agile Project Manager needs to adopt in order to support and enable self-organising teams. The basics of DSDM (see Section 14.3) also form the basis of AgilePM.

AgilePM has also been defined in such a way that it can be used to provide a 'project' wrapper for other Agile approaches. For example, where Scrum is the preferred Agile choice at team level, but where an overarching project framework is still needed. DSDM has also published a pocket book describing a customisation of AgilePM framework customised specifically to meet the needs of a Scrum project.

14.5 KANBAN

The Kanban method (Anderson, 2010) is an approach to continuously improving service delivery that emphasises the smooth, fast flow of work. Organisationally Kanban is used for all creative and knowledge work service delivery and workflows. It has been used in video editing, advertising, recruitment, legal case processes, finance, sales, marketing, community development and management, etc. This section will concentrate on Kanban within delivery and maintenance of IT systems.

Kanban is not an Agile software development method (or process) or a software engineering methodology; it is an alternative path to agility or a method for improving service delivery agility aimed at improving business resilience and fitness for purpose. It's also been described as a method for improving organisational agility as opposed to product development agility with first generation Agile methods.

Kanban does not prescribe specific roles or process steps as it is built on the concept of evolutionary change. Start by understanding how the current software delivery system works. When the current flow of work is visualised and measured, improve it one step at a time. Continue to do this forever.

14.5.1 Six core Kanban practices

The six core practices are;

- Visualise the work;
- Limit work-in-progress (WIP);
- Make policies explicit;
- Measure and manage flow;
- Implement feedback loops;
- Improve collaboratively, evolve experimentally.

These are presented in further detail in the following subsections.

14.5.1.1 Visualise the work, the workflow and business risks

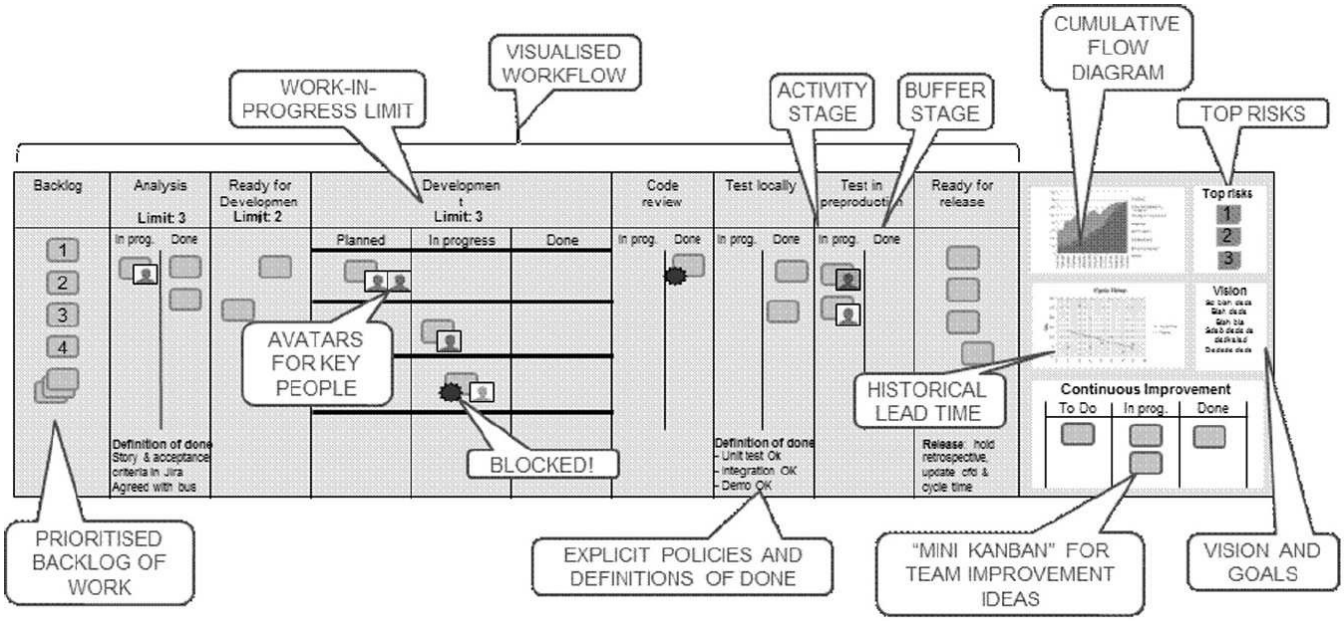
The concept of 'inventory' in a software delivery system is tricky – it is effectively invisible since we are dealing with the flow of ideas. Visualising each step in the value chain from vague idea to released software is a prerequisite for effective management of the end-to-end delivery process. This is normally done on a physical board with Post-It® notes or similar to represent the work (see Figure

14.6). Work flows across the board from left to right.

14.5.1.2 Limit work-in-progress (WIP)

Limiting the amount of work items being worked on at any one time is a counter-intuitive approach to improving the flow of work (‘work items’ in this context means customer-valued (or requested) work and not tasks). Kanban teams are interested in tracking the flow of value and not the effort expended. Kanban is about managing customer-valued work rather than managing people. The demands of the customer – what they want and when they need it – determine how people organise their work to deliver against these expectations. By limiting the amount of work-in-progress at particular process steps, work items are held upstream until another work item is finished, capacity then becomes available and they can be pulled into the process. A kind of ‘one in, one out’ policy – similar to that operated by bouncers at busy nightclubs!

Figure 14.6 Typical Kanban board



Lowering WIP reduces coordination costs (less to coordinate!), reduces multi-tasking and increases focus. The process becomes more responsive to unplanned external events (e.g. ‘we have a new urgent requirement’) and process failures (e.g. ‘we found in our final testing that the new architecture is not going to work.’). Less WIP also results in reduced lead times (time it takes for work items to flow through the process) since we have a small number of fast flowing items instead of a large number of slow moving items. Average lead time is directly related to WIP according to Little’s law (Little and Graves, 2008).

Perhaps less obvious is that gradually lowering the WIP limits drives collaborative improvement of the whole process. Restricting WIP will cause some process steps (those with overcapacity) to be starved of work. This ‘pain’ prompts the operators and the management of these process steps to investigate what is happening upstream and downstream (seeing as this is conveniently visualised, this should not be too difficult). This typically results in a process redesign or optimisation to improve the flow between these process steps. Setting appropriate WIP levels is

somewhat of a ‘black art’ – set them too high and they have no effect, set them too low and they choke off the flow of work. Experimenting with WIP limits is encouraged, as it is relatively easy to reverse any changes to the WIP limits.

14.5.1.3 Make policies explicit

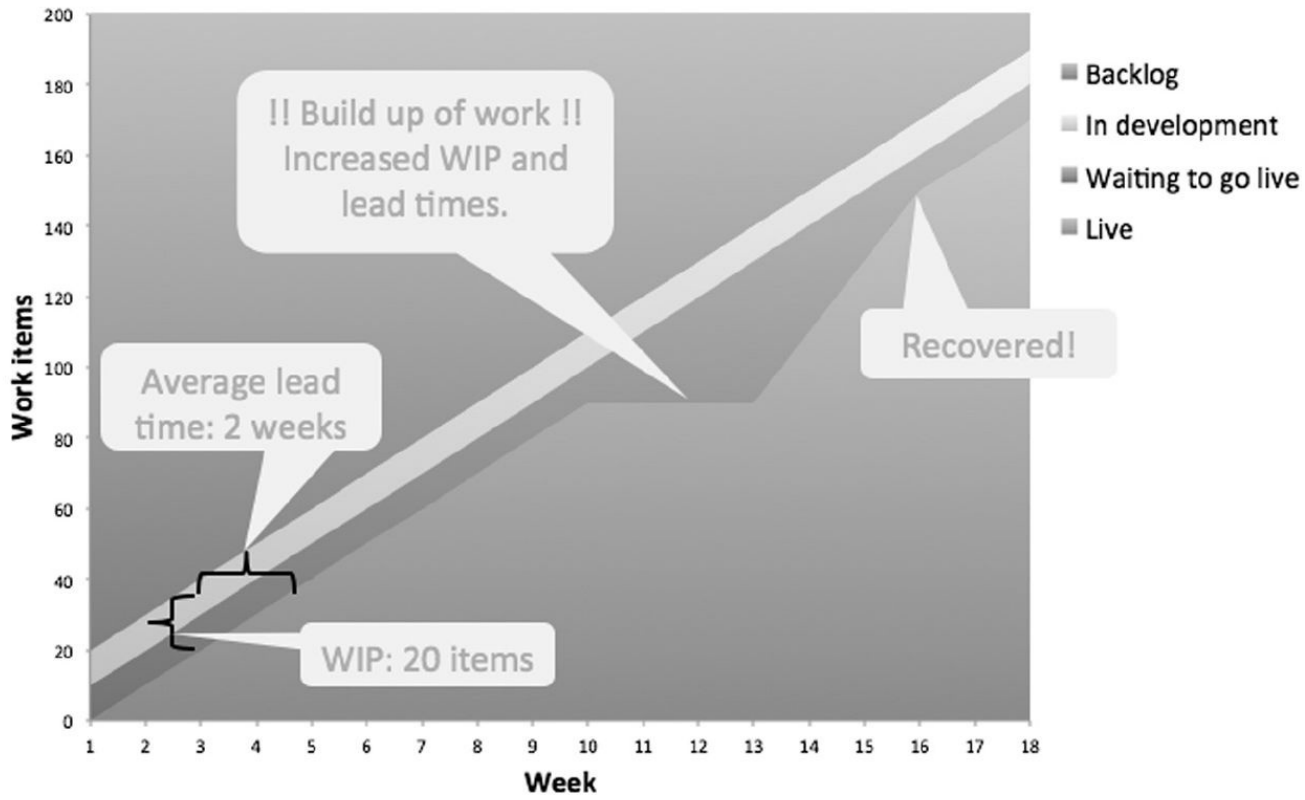
All management, risk management and process policies that apply to the process must be documented. For example, the team may produce checklists of what needs to have happened in order to consider a specific step in the process as complete or they may document how particular decisions are to be made.

14.5.1.4 Manage flow

Transitions between process steps in the workflow are monitored and measured. This gives a historical picture of the flow of work, lead times etc. Analysing this gives insights into opportunities for improvement and also determines whether any changes previously made have actually resulted in an improvement to the process.

Cumulative flow diagrams (Figure 14.7) are powerful tools for visualising the history of the flow of work. WIP levels on any given date can be read off the vertical scale (see week 3 in Figure 14.7). Less obviously, average lead times can be read off the horizontal scale (2 weeks for work items begun in week 3). The slope of a line in a cumulative flow diagram is the rate at which work is transitioning from one process step to the next – the steeper the line, the greater the rate of transition.

Figure 14.7 Cumulative flow diagram



14.5.1.5 Implement feedback loops

Kanban encourages the use of feedback loops at all levels to facilitate learning about the process and about the effect of any changes that have been made to it.

14.5.1.6 Improve collaboratively, evolve experimentally

A significant part of the Kanban method is its emphasis on the creation of a culture to support continuous change. In particular, a ‘Kaizen’ (Womack, Jones and Roos, 2007) culture in which continuous improvement is seen as everyone’s job at all levels within the organisation and all feel empowered to make changes.

14.5.2 Kanban models

Kanban encourages the use of models to identify changes that, if applied, will likely result in an improvement. Common Kanban models include:

- The theory of profound knowledge or the PDCA/PDSA cycle (the most common model; Deming, n.d.).
- The real option theory (Matts, 2013) and related concepts from financial risk management such as market liquidity.
- Sense-making techniques for clustering assignable cause variations are now being synthesised with event-driven risk management from the project management body of knowledge or PMBoK (PMI, 2013).
- The theory of constraints (Goldratt and Cox, 1984).
- Transaction, coordination costs and information arrival rates (and Shannon’s information theory) is also used (Shannon, 1948).

14.5.3 Kanban origins

Kanban was adapted from the Lean manufacturing (Liker, 2004) approach of the same name by David J. Anderson in 2007 (Anderson, 2010). As David Anderson himself says: ‘What Kanban first and foremost does is serve as a catalyst to introduce Lean ideas into software delivery systems’ (Anderson, 2010).

The word ‘Kanban’ itself is a Japanese word used originally by Toyota to mean ‘signal card’. The ‘signal card’ is physical card in the Toyota plant that signalled when to pull more inventory from a supply step or storage location.

14.5.4 Kanban and Scrum

Kanban is often contrasted with Scrum (see Section 14.2).

Table 14.1 Kanban/Scrum comparison

Kanban	Scrum
An approach to improving service delivery using evolutionary improvement.	An approach to complex product development.
No roles or processes defined.	Standard roles and processes clearly defined.
Evolutionary. Start with what already exists and iterate. Respect what is.	Revolutionary. Do it like Scrum says. Doesn't matter what has gone before.
WIP limited by a 'pull' system based on work orders or requirements. Limit the work and let time vary.	WIP never exceeds the amount that can be completed by the team in a sprint. Limit the time and vary the work to fit the defined time period.

Organisations sometimes combine Kanban and Scrum, although either framework will work perfectly well on its own. There are several ways this can be valuable:

- 'Scrumban' (Ladas, 2009) (Kanban within a Scrum team) – Scrum teams use Kanban to manage the work they do within a sprint.
- Enterprise (Kanban above Scrum team) – Scrum teams may contribute to larger programmes of work involving multiple teams. These larger programmes apply Kanban at a higher level to manage some or all of the work. The portfolio level in SAFe (See Section 14.8) is a good example of this.
- End-to-end process (Kanban upstream and downstream of Scrum team) – A full Scrum implementation implies there is no downstream or upstream activity outside the team – everybody who is required to deliver value should be within the boundaries of the team. In practice, teams rarely start like this. Kanban can be helpful in visualising and addressing this situation both upstream (requirements analysis, architectural design, etc.) and downstream (release management, integration testing, documentation, etc.) of the Scrum team.
- Adopt Kanban for work that does not fit Scrum – Many teams work on items that are unsuitable for Scrum. Some will not be able to create a 'useable and potentially shippable product increment' within a month (e.g. teams working on large legacy software systems with lots of manual tests). Other teams will experience so much requirement volatility that they are unable to make a plan for a whole sprint (e.g. reactive support work). Yet other teams do not have a 'product increment' as their primary output (e.g. DevOps team, enterprise architect team, senior management team, etc.). In these contexts, Kanban without Scrum is a good fit.
- Some Agile teams adopt Kanban without Scrum because they are unable to gain sufficient internal alignment to implement the major organisational and

procedural changes required for effective Scrum.

14.6 LEAN SOFTWARE DEVELOPMENT

Lean software development originated in 2003 with the work of Mary and Tom Poppendieck (Poppendieck, 2003). It is based on adapting ideas from Lean manufacturing to a software development context. This approach has been enthusiastically embraced by the Agile community. Lean principles often provide an explanation for why many Agile practices work, based on proven evidence and also provide a framework for improving and scaling an Agile approach. In fact, the boundary between Lean and Agile has become blurred over time.

The original Lean manufacturing approach was created by Toyota (Womack, Jones and Roos, 2007) and focuses on the elimination of non-value-adding activities (waste) in the manufacturing process. Some adjustment to this approach is needed as the nature of software development is fundamentally different from manufacturing in two key ways:

- Manufacturing is a repetitive process. Software development is not repetitive as both requirements and solutions are unique. They also typically build on previous requirements/solutions. This places greater emphasis on learning – both about how to build a solution and whether it is valuable or not. In particular, learning is achieved through fast feedback.
- Both manufacturing and software development processes are able to influence the cost and quality of their respective outputs, but only software development can make a significant difference to the value of the product in the customer's eyes by discovering valuable stories that were not thought of up front and avoiding developing stories that have no value. In fact, a Lean software approach suggests that this is likely the biggest win in terms of improvement efforts. As such, the original Lean manufacturing focus of elimination of non-value (waste) is typically recast to the broader 'maximise value' in order to include the possibility of value creation.

The full Lean software development approach consists of seven core principles (Poppendieck and Poppendieck, 2007), supported by 22 tools. Mary Poppendieck boils down Lean software development as it applies to a team (Poppendieck, 2014) to five key questions:

- Is the team a whole team – composed of everyone necessary to deliver value to the ultimate customer?
- Does everyone on the team understand what customers really value?
- Is the team focused on delivering small increments of real value to end customers?
- Does the team reliably and repeatedly deliver on its promises?
- Is there a leader who understands and cares deeply about the customers and their problems and a leader who understands and cares deeply about the

technical integrity of the product?

14.6.1 Seven Lean software development principles

14.6.1.1 Principle 1: Eliminate waste

The three biggest wastes in software development are:

- **Extra stories** A process is needed for developing just those 20 per cent of the stories that give 80 per cent of the value.
- **Churn** If there is requirements churn, specification is happening early. If there are test and fix cycles, the testing is too late.
- **Crossing boundaries** Organisational boundaries can increase costs by 25 per cent or more. They create buffers that slow down response time and interfere with communication.

14.6.1.2 Principle 2: Build in quality

If defects are routinely found in the verification process, the development process is defective.

- Mistake-proofing code with test-driven development. Writing executable specifications instead of requirements.
- Not building legacy code. Legacy code is code that lacks automated unit and acceptance tests.

14.6.1.3 Principle 3: Create knowledge

Planning is useful, learning essential.

- Use the scientific method. Teach teams to establish hypotheses, conduct many rapid experiments, create concise documentation and implement the best alternative.
- Standards exist to be challenged and improved. Embody the current best-known practices in standards that are always followed while actively encouraging everyone to challenge and change these standards.
- Predictable performance is driven by feedback. Don’t guess about the future and call it a plan; use feedback based on reality to develop the capacity to rapidly respond to the future as it unfolds.

14.6.1.4 Principle 4: Defer commitment

Abolish the idea that it is a good idea to start development with a complete specification.

- Break dependencies. System architecture should support the addition of any PBI at any time.
- Maintain options. Think of code as an experiment – make it change-tolerant.

- Schedule irreversible decisions at the last responsible moment. Learn as much as possible before making irreversible decisions

14.6.1.5 Principle 5: Deliver fast

Queues (points in a delivery process where the process stops whilst waiting for something, like a sign-off) are buffers between organisations that slow things down.

- Rapid delivery, high quality and low cost are fully compatible. Companies that compete on the basis of efficient, rapid, value chains have significant cost advantage, deliver superior quality, and are most attuned to their customers' needs.
- Queuing theory applies to development, not just servers. Focusing on utilisation creates traffic jams that actually reduce utilisation. Drive down cycle time with small batches and restrict work-in-progress where required.
- Limit work to capacity. Establish a reliable, repeatable velocity with iterative development. Aggressively limit the size of work in progress to optimise delivery flow.

14.6.1.6 Principle 6: Respect people

Engaged, thinking people provide the most sustainable competitive advantage.

- Teams thrive on pride, commitment, trust and applause. What makes a team? Members are mutually committed to achieving a common goal.
- Provide effective leadership. Effective teams have effective leaders who bring out the best in the team.
- Respect partners. Allegiance to the joint venture must never create a conflict of interest, the venture must be beneficial to both parties.

14.6.1.7 Principle 7: Optimise the whole

Brilliant products emerge from a unique combination of opportunity and technology.

- Focus on the entire value stream. From concept to cash. From customer request to deployed software, focus on the whole delivery process from start to end, not just one part of the process.
- Deliver a complete product. Develop a complete product, not just software; this may include documentation, business change, etc. Complete products are built by complete teams.
- Measure up. Measure process capability with cycle time. Measure team performance with delivered business value. Measure customer satisfaction with a net promoter score (a customer loyalty metric; Reichheld 2003).

14.6.2 Twenty-two Lean software development tools

The seven core principles are supported by 22 tools.

- **Seeing waste** Some wastes are obvious, defects for example. Some are more subtle: extra stories, waiting, task switching.
- **Value stream mapping** Map out the flow of a requirement from idea to rollout. This helps identify queues, waiting, excessive hand-offs, non-value-add steps and so on.
- **Feedback** Increase the speed and the quality of feedback at many levels to improve process quality.
- **Iterations** Deliver early and often to get feedback on the product and its value.
- **Synchronisation** Integrate software frequently to avoid surprises and rework.
- **Set-based development** Work on multiple options or let the solution emerge.
- **Options thinking** Keeping options open has a value – delay decisions if possible until more information is available.
- **Last responsible moment** This is the point at which, if a decision is delayed any further, an important option will be eliminated. Delay the decision no further than this.
- **Making decisions** Devolve decision to the team level as far as possible and be guided by simple decision-making rules (e.g. agree rules that define a clear prioritisation method for stories in the backlog with all stakeholders).
- **Pull systems** Allow teams to pull in work when they have capacity. Do not push the work to them as this increases the amount of work-in-progress without increasing throughput.
- **Queuing theory** Identify, measure and reduce queues of partially complete work in the development process. There are numerous ways to do this, however, measuring cost of delay is a good start point (see Section 10.1).
- **Cost of delay** Build an economic model that puts a price on speed of delivery so it can be traded off against other factors like risk, cost and so on.
- **Self-determination** Allow teams to decide how they organise their own work.
- **Motivation** Team motivation comes from an achievable purpose, access to the customer and a freedom to make their own commitments in an environment in which mistakes are opportunities to learn.
- **Leadership** Most successful initiatives have a product champion and a master developer who care passionately about the customer and the integrity of the product respectively.
- **Expertise** Share expertise and enforce standards by the Agile lead (see Section 6.3) facilitating the team to become self-organising and to work in a collaborative way.
- **Perceived integrity** Align all activities to customer value so that it is clear why things are happening.
- **Conceptual integrity** Ensure that components/interfaces of the product work

together as a smooth cohesive whole.

- **Refactoring** Essential to maintain the health of the product as not everything can be foreseen up front at design time.
- **Testing** Automate testing to speed up the feedback loop.
- **Measurements** Make problems and progress visible.
- **Contracts** Avoid fixing scope up front in a contract.

14.7 LEAN START-UP

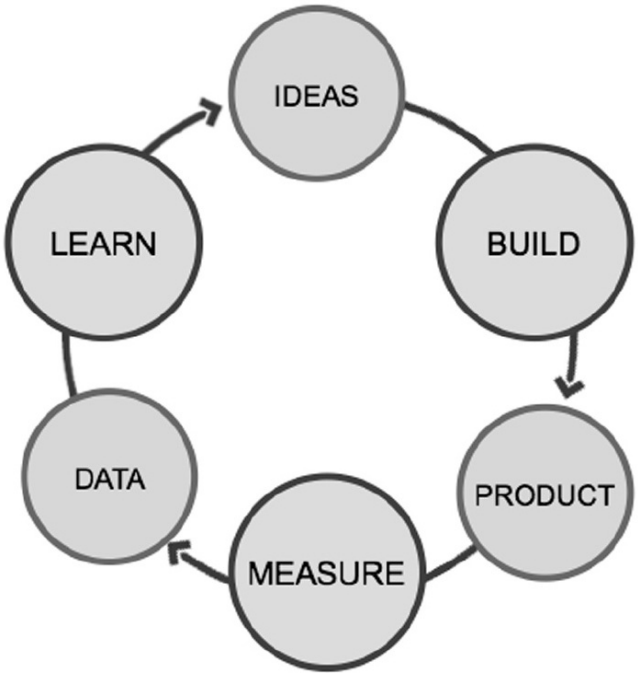
The Lean start-up approach began with a book of the same name (Ries, 2011) and is now a worldwide movement. It is rooted in case studies and the experiences of its creator, Eric Ries. Note that the name can be misleading – running your own company in a Silicon Valley garage is not required to be considered a start-up! In fact, 'anybody who is developing a new product … under conditions of extreme uncertainty is a start-up' (Ries, 2011: 27). Entrepreneurs are everywhere.

When used in an Agile approach, it is most useful to the product management of brand new products. Lean start-up is not about executing a business model (existing companies do this), but to find a business model. A business model has to answer the following questions:

- How do customers discover the product?
- How do customers get value from the product?
- How is sufficient revenue generated by this for the company to make a profit?

It has been observed that most successful start-ups go from failure to failure before finding a business model that works. The experiments are a series of build–measure–learn feedback loops (see Figure 14.8). An entrepreneur starts with an idea for a business model. They then build a product (or part of a product), which is then put in front of real customers and the results are measured. Learning from these measurements then generates further ideas about how they adjust the business model.

Figure 14.8 The build–measure–learn feedback loop



The entrepreneur's role is to maximise the chance of finding a successful business model before running out of funding. This is done by making build–measure–learn loops quick, cheap and effective. Managing this process is what entrepreneurship is all about and the unit of progress is learning.

At the start of the build–measure–learn loop, the entrepreneur has a number of 'leap-of-faith assumptions', which need to be tested. These cover at least:

- **Value hypothesis** Who will see value in the new product or service?
- **Growth hypothesis** How will customers discover the new product or service?

A business model canvas (Blank, 2013) is a common format used to capture on one page all the hypotheses that need testing. It may contain (non-exhaustive list):

- Key partners (suppliers, delivery partners, who does what).
- Key tasks (to get to start-up).
- Key resources and people (equipment, offices, people etc. required).
- Value proposition and MMFS(s) (why should customers buy? Minimum Marketable Feature Set, services/products).
- Finding customers (how to attract customers – social media/campaigns?).
- Market segments (who are being targeted and for what reason(s)?).
- Channels to market (how will we get the right message to those we are targeting?).
- Key costs (over basic timeline).
- Key revenues (over basic timeline).

Given the extreme uncertainty in these hypotheses, the value of getting customer feedback about them far outweighs the benefits of keeping this new product development activity secret from the competition. Entrepreneurs must get out of

the building and interact with customers to gauge their reactions – it is the only way to validate their learning.

Hypotheses need to be small enough so they can be tested quickly and the metrics resulting from an experiment need to be actionable. It is important to avoid so-called vanity metrics which look interesting but don't really measure anything that can confirm or deny the hypotheses (e.g. 'number of page views on site' looks good, however, the page may be alienating people or page creating views costs too much).

Example hypotheses from Votizen, a US start-up

Product Concept:

Social network of verified voters, a place where people passionate about civic causes can interact.

Hypotheses:

- Customers are interested enough to sign up [Registration].
- Votizen can verify them as registered voters [Activation].
- Verified voters engage with the network over time [Retention].
- Engaged customers tell their friends about the service [Referral].

Once the hypotheses are identified, Lean start-up recommends applying Lean and Agile development practices to create the minimum viable product (MVP). The MVP is the simplest version of the product that enables a full turn of the build–measure–learn loop.

Once an experiment is run, the moment comes for a decision. Pivot or persevere – i.e. pivot to a different set of hypotheses or persevere and run further experiments tuning the existing product? Pivoting can take many forms: change of customers segment, channel, zooming in on fewer stories, addressing an adjacent customer need that has just been uncovered and so on.

14.8 SCALED AGILE FRAMEWORK (SAFE)

Authors' note: Scaling Agile is a topic that causes much discussion in the Agile community as of 2014. Most of the frameworks described in Part 4 of this book can be scaled on their own. However, the majority of them do not (purposefully, because they are non-prescriptive frameworks) explain how they are scaled. There are also many other frameworks specifically focused on scaling Agile; examples are LESS (Large Scale Scrum; Larman and Vodde, 2013) and DAD (Disciplined Agile Delivery; Ambler and Lines, 2012). The authors think that 'SAFe' is currently the most fully featured, discussed and implemented Agile scaling framework, which is why we have included it in this book.

This section describes SAFe version 3.0.

The Scaled Agile Framework (SAFe) (Leffingwell, 2011–14) is a scaled approach

to Agile adoption. Officially launched in 2012, it draws on the experiences of its creator, Dean Leffingwell, and his co-collaborators as to 'effective practices that worked for us' when they found Scrum (see Section 14.2) and eXtreme Programming (see Section 14.1) insufficient in addressing the problems faced by large software organisations.

SAFe is a mix of original work and a container for several existing Agile approaches. The SAFe framework provides:

- A process model that covers the highest and the lowest level in the enterprise. Of particular importance is:
    - A mid-level planning cycle (the program increment, or PI) every 4–6 sprints that covers the activities of 5–12 Scrum teams (an Agile program).
    - Funding of these stable long-term programs, which are aligned to a flow of value to the customer (a value stream).
- Associated Agile values and practices. Many of these practices summarise or simply refer to other Lean or Agile approaches. SAFe endorses or makes reference to: Scrum (see Section 14.2), eXtreme Programming (see Section 14.1), Kanban (see Section 14.5), Lean thinking and Lean product development flow (see Section 14.6), and the Agile Manifesto (see Section 1.2).
- Four core values: code quality, alignment, program execution and transparency.

14.8.1 SAFe process model

SAFe is organised around three layers:

- Teams, who adopt Scrum (or Scrumban) and eXtreme Programming.
- Programs, each of which contains 5–12 teams working towards a common goal.
- Portfolio for funding and coordinating programs.

14.8.1.1 Teams

Figure 14.9 shows that SAFe ScrumXP Teams (Agile teams) power SAFe development and use Scrum (or Scrumban) and eXtreme Programming (XP) as the basis of their work (Leffingwell, 2011–14). There are some adjustments to be made to these standard Agile frameworks, including an understanding of the team's roles in the program, and an emphasis on flow and limiting work-in-progess.

14.8.1.2 Program

In Figure 14.10 the 'Agile release train' (also known as an Agile program) is the primary organisational, operational and value delivery construct. Five to twelve SAFe scrumXP teams form each Agile release train in a value stream. This team-

of-teams operates on a planning cycle (program increment) of 8 to 12 weeks.

SAFe describes a ‘develop on cadence, release on demand’ approach. The development cadence is used to help manage the intrinsic variability of research and development. Releasing occurs on demand, either synchronously or asynchronously with the PI cycle. SAFe makes no recommendations as to release frequency, and leaves that to the discretion of the individual release trains.

SAFe defines nine roles at the program level:

- **Product management** Prioritises program backlog. Owns and communicates product vision and roadmap.
- **Release train engineer** Drives program level continuous improvement. Facilitates PI release (a kind of ‘ScrumMaster’ (see Section 14.2) for the Agile release train).
- **Business owner** Senior management responsible (ultimately) for value delivery. Participates in release planning and release inspect and adapt (see below).
- **System architect** Helps break down system-level stories. Liaises with enterprise architecture.
- **User Interface team** (UX) Creates a consistent user experience.
- **System team** Provides process and tools to integrate and evaluate early and often. Enables system-level continuous integration.
- **DevOps** Characterises the skills needed by the programs to build and improve a rapid deployment pipeline.
- **Release management** Synchronises releases with other programs and stakeholders.
- **Shared resources** Provide assistance in specialist areas not normally found in the teams.

Figure 14.9 SAFe team level

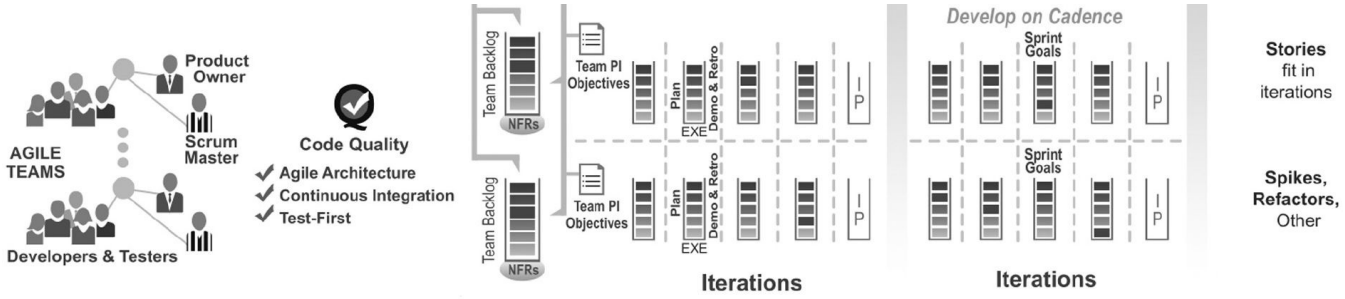
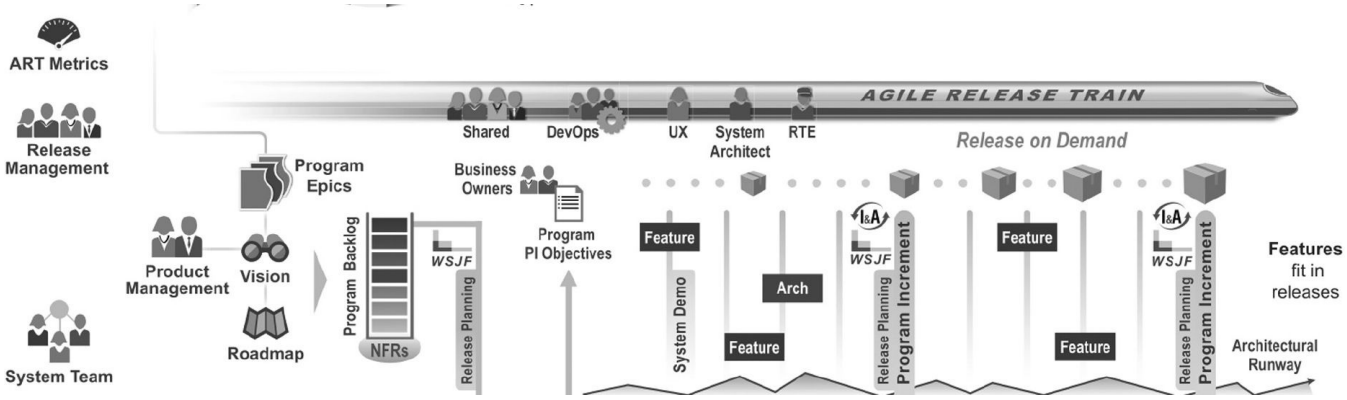


Figure 14.10 SAFe program level



Each program increment (PI) begins with a 2 day 'all hands' planning session, which everybody in the program attends. It is at this event that the overall PI objectives are identified along with team-specific PI objectives, and a release plan is made to achieve them over the next four to six sprints. A vote of confidence/commitment in the release plan that has been defined is also held.

Once the train is underway, coordination between teams is achieved, in part, through a standard Scrum-of-Scrums meeting (see Section 14.2), facilitated by the release train engineer. A system-level demonstration is given to relevant stakeholders at the end of each sprint to review progress. The final sprint in the release plan cycle may be an innovation and planning (IP) sprint. This provides time for:

- Innovation, exploration and out-of-the-box thinking; and
- Planning the next PSI release.

Scheduling IP sprints at the end of each program increment can also serve as a schedule buffer if planned development overruns.

An inspect and adapt workshop (I&A) is at the end of each program increment. The I&A is to a SAFe program as a sprint review and retrospective are to a Scrum team (see Section 14.2), that is, an opportunity for reflection and adjustment, but at the level of the program, where most of the larger, systemic impediments occur.

14.8.1.3 Portfolio

The SAFe portfolio layer (see Figure 14.11) is responsible for:

- Coordinating larger initiatives (epics) that require implementation across multiple release trains. These are managed in the portfolio backlog.
- Funding of Agile release trains.

The portfolio backlog contains epics that cross multiple release trains. These can be either customer-facing (business epics) or technology (architectural epics) initiatives. To get into the backlog, initiatives need analysis and approval. The work at this level is managed using a Kanban system (See Section 14.5).

Every Agile release train is aligned within a value stream (i.e. a significant flow of value to the customer) and is funded by an associated budget. Budgets are determined by the current business context and the evolving portfolio strategic

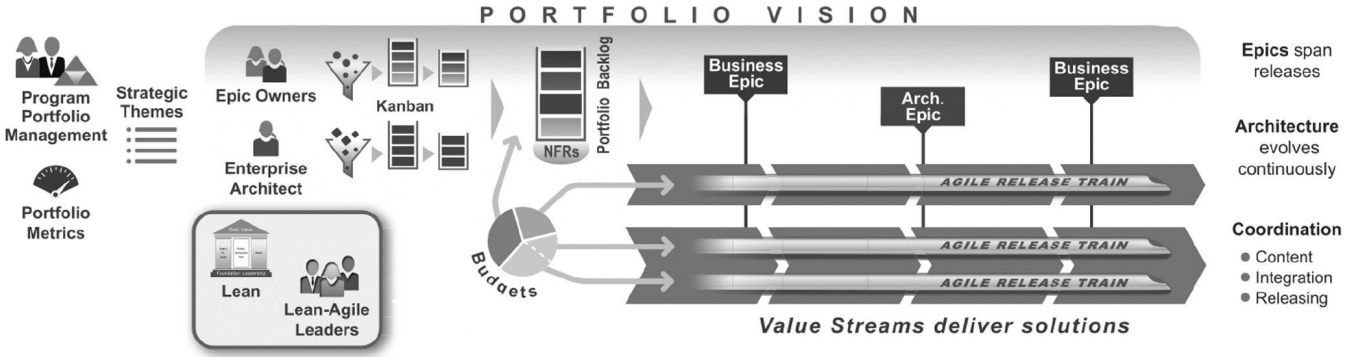
themes.

Roles in the SAFe portfolio layer are more loosely defined than they are in the other two layers of SAFe. They include program portfolio management, enterprise architecture and epic owners.

14.8.2 SAFe Agile architecture

SAFe contends that **emergent architecture** (design grown incrementally), while effective for the team's local concerns, is not sufficient at scale. Some intentional architecture is required (design done up front). SAFe aims to provide just enough of this up-front architecture just in time to enable teams to progress effectively. Architectural features that are built just in time to enable business features result in software code known as the architectural runway.

Figure 14.11 SAFe portfolio level



Non-functional requirements (reliability, scalability, maintainability and so on) are key architectural concerns in SAFe. They are most typically associated with program-level backlogs, either as a single backlog item or a constraint on all backlog items (e.g., 'web pages load in under 2 s').